Part NINE
JDBC and Localization

Chapter TWENTY-NINE
JDBC API

Exam Objectives

Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations.
Identify the components required to connect to a database using the DriverManager class including the JDBC URL.
Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections.

Introduction

The Java Database Connectivity API (or JDBC) defines how we can access a relational database. You can find its classes and interfaces in the `java.sql` package.

The latest version of this API, included in Java 8, is JDBC 4.2.

The main benefit of using JDBC is that it hides the implementation details between databases by using a set of interfaces so you don't have to write different code for accessing different databases.

This chapter assumes you know how to use SQL to select, insert, update, and delete records from a database entity.

Also, for this chapter, it's recommended (although not required) to have access to a database to do some practices and test some concepts, however, it won't teach how to install or use one.

If you don't have a database installed (or access to one), perhaps the easiest way is to use `JavaDB`, a database that is shipped with Java that can be used as an embedded or network server database. You can find it in the db directory of the JDK installation.

For the exam, you are expected to know the main interfaces of JDBC, how to connect to a database, read results of a query, and perform some operations such as inserts or updates.

JDBC Interfaces

When using JDBC, you work with interfaces rather than implementations. Those implementations come from a JDBC driver.

A driver is just a JAR file with classes that know how to talk to a specific database. For example, for MySQL, there's a `mysql-connector-java-XXX.jar`, where XXX is the version of the database.

But you don't need to know how the classes inside the driver are implemented or named, you just have to know the four main interfaces that they have to implement:

- `java.sql.Driver`
  Every JDBC driver must implement this interface to know how to connect to the database.
- `java.sql.Connection`
  The implementation provides methods for getting information about the database, create statements, and managing connections.
- `java.sql.Statement`
  The implementation is used to execute SQL statements and to return results.
- `java.sql.ResultSet`
  The implementation is used for retrieving and updating the results of a query.

In addition to these interfaces, an important class is `java.sql.DriverManager`, which keeps track of the loaded JDBC drivers and gets the actual connection to the database.

Connecting to a Database

First, to work with a database, you need to connect to it. This is done using the `DriverManager` class.

Before attempting to establish a connection, the `DriverManager` class has to load and register any implementations of `java.sql.Driver` (which know how to establish the connection).

Maybe you've seen in some program a line like this:

```
Class.forName("com.database.Driver")
```

This loads the Driver implementation so `DriverManager` can register the driver. However, this is no longer required since JDBC 4.0, because `DriverManager` automatically loads any JDBC 4.0 driver in the classpath.

Once the drivers are loaded, you can connect to a database with the `static` method `DriverManager.getConnection()`:

```
Connection getConnection(String url)
Connection getConnection(String url,
                         Properties info)
Connection getConnection(String url,
                         String user,
                         String passw)
```

This method will look through the registered drivers to see if it can find one that can make a connection using the given URL.

The URL varies depending on the database used. However, they have three parts in common:

JDBC URL Format

1. Protocol (always the same)

2. Subprotocol (most of the time the name of the database/type of the driver)

jdbc:mysql://host:3306/db

3. Database specific connection properties (most of the time the location of the database with format: //SERVER:HOST/DATABASE_NAME)

- The first part is always the same, *jdbc*.
- The second part, most of the time, is the name of the database and/or the type of the driver, like *mysql*, *postgresql*, *oracle:thin*.
- The last part varies according to the database, but most of the time, it contains the name (or IP) of the host, the port and the name of the database you're connecting to, like *//192.168.0.1:3306/db1*.

Here are some URL examples:

```
jdbc:postgresql://localhost/test
jdbc:sqlserver://localhost\SQLEXPRESS;databasename=db
jdbc:derby:db;create=true
```

We can connect to the database by calling this method to get a `Connection` object:

```
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost/db");
```

If you need to pass a user or password for authentication, we can use:

```
Properties props = new Properties();
props.put("user", "db_user");
props.put("password", "db_p4assw0rd");
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost/db",
                                props);
```

Or:

```
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost/db",
                                "db_user",
                                "db_p4assw0rd");
```

Before your program finishes, you need to close the connection (otherwise you can run out of connections).

Luckily, `Connection` implements `AutoCloseable`, which means you can use a `try-with-resources` to automatically close the connection:

```
String url = "jdbc:mysql://localhost/db";
String user = "db_user";
String passw = " db_p4assw0rd";

try(Connection con =
        DriverManager.getConnection(url, user, passw)) {
    // Database operations
} catch(SQLException e) {
    System.out.format("%d-%s-%s",
                      e.getErrorCode(),
                      e.getSQLState(),
                      e.getMessage());
}
```

An `SQLException` is thrown whenever there's an error in JDBC (like when the driver to connect to the database is not found in the classpath) and many methods of the JDBC API throw it, so it has to be caught (or retrieved in the case of suppressed exceptions).

As this exception is used for many errors, to know what went wrong, you have to use `getMessage()`, which returns a description of the error, `getSQLState()` that returns a standard error code, or `getErrorCode()`, which returns the database-specific code for the error.

Now that we have a `Connection` object, we can execute some SQL statements.

Executing Queries

You a need a `Statement` object to execute queries and perform database operations.

There are three `Statement` interfaces:

- `java.sql.Statement`
  Represents simple SQL statements to the database, without parameters.
- `java.sql.PreparedStatement`
  Represents precompiled SQL statements to execute statements multiple times efficiently. It can accept input parameters.
- `java.sql.CallableStatement`
  Used to execute stored procedures. It can accept input as well as output parameters.

In practice, `PreparedStatement` is the one often used, but for the exam, you only need to know `Statement`.

You can get a `Statement` from a `Connection` object using the `createStatement()` method:

```
Statement createStatement()
Statement createStatement(int resultSetType,
                          int resultSetConcurrency)
```

When creating a `Statement`, you can define the type of the result set and its concurrency mode.

There are three types of result sets:

- `ResultSet.TYPE_FORWARD_ONLY`
  This is the default type. When specified, you can only go once through the results and in the order they were retrieved.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`
  When specified, you can go both forward and backward through the results and to a particular position in the result set.
- `ResultSet.TYPE_SCROLL_SENSITIVE`
  When specified, you can also go forward, backward and to a particular position in the result set, but you will always see the latest changes to the data while using it, in contrast with `TYPE_SCROLL_INSENSITIVE`, which it's not "sensitive" to changes to the data.

In practice, most drivers don't support `TYPE_SCROLL_SENSITIVE`. If you ask for it and is not available, you will get either `TYPE_FORWARD_ONLY` or (more likely) `TYPE_SCROLL_INSENSITIVE`.

There are two concurrency modes:

- `ResultSet.CONCUR_READ_ONLY`
  This is the default mode. When specified, you can't update (using an `INSERT`, `UPDATE`, or `DELETE` statement) a result set.
- `ResultSet.CONCUR_UPDATABLE`
  It indicates that the result set can be updated.

If you ask for a `CONCUR_UPDATABLE` mode and your driver doesn't support it, you can get a `CONCUR_READ_ONLY` mode.

Most of the time, the default values are used:

```
Statement stmt = con.createStatement();
```

Now that we have a `Statement` object, we have three methods at our disposal to execute SQL commands:

Method	Supported SQL statements	Return type

`execute()`	`SELECT`
	`INSERT`
	`UPDATE`
	`DELETE`
	`CREATE`
		`boolean` (`true` for `SELECT`, `false` for the rest)

`executeQuery()`	`SELECT`	`ResultSet`

`executeUpdate()`	`INSERT`
	`UPDATE`
	`DELETE`
	`CREATE`
		Number of affected rows (zero for `CREATE`)

A `Statement` object has to be closed, but like `Connection`, it implements `AutoCloseable` so it can be used with a `try-with-resources` also:

```
try(Connection con =
      DriverManager.getConnection(url, user, passw);
      Statement stmt = con.createStatement()) {
   boolean hasResults = stmt.execute("SELECT * FROM user");
   if(hasResults) {
      // To retrieve the object with the results
      ResultSet rs = stmt.getResultSet()
   } else {
```

```
      // To get the number of affected rows
      int affectedRows = stmt.getUpdateCount();
   }
   ResultSet rs = stmt.executeQuery("SELECT * FROM user");
   stmt.executeUpdate("INSERT INTO user(id, name) "
        + "VALUES(1, 'George')"); // Returns 1
   stmt.executeUpdate("UPDATE user SET name='Joe' "
        + "WHERE id = 1"); // Returns 1
   stmt.executeUpdate("DELETE FROM user "
        + "WHERE id = 1"); // Returns 1
} catch(SQLException e) {
   e.printStackTrace();
}
```

Choose the correct `execute()` method based on the type of SQL statement you're using, because if you use the wrong method, an `SQLException` will be thrown.

When you use a `SELECT` statement, you can read the result through a `ResultSet` object, which we'll review next.

Reading Results

A `ResultSet` object is used to read the results of a query in a tabular format (rows containing the columns specified).

This object keeps a cursor that points to the current row and you can only read one row at a time.

In the beginning, the cursor is just before the first row. Calling the `next()` method will advance the cursor one position and return `true` if there's data, or `false` if there isn't any. This way, you can iterate in a loop over the entire result set.

Like `Connection` and `Statement`, a `ResultSet` object needs to be closed too and also implements `AutoCloseable`:

```
try(Connection con =
      DriverManager.getConnection(url, user, passw);
      Statement stmt = con.createStatement();
      ResultSet rs =
         stmt.executeQuery("SELECT * FROM user")) {
   while(rs.next()) {
      // Read row
   }
} catch(SQLException e) {
   e.printStackTrace();
}
```

It's a good practice to close these resources in this way, but it's not required. Here are the rules for closing JDBC resources:

- The `ResultSet` object is closed first, then the `Statement` object, then the `Connection` object.
- A `ResultSet` is automatically closed when another `ResultSet` is executed from the same `Statement` object.
- Closing a `Statement` also closes the `ResultSet`.
- Closing a `Connection` also closes the `Statement` and `ResultSet` objects.

If the query doesn't return results, a `ResultSet` object is still returned (although `next()` will return `false` at the first call).

Remember, `next()` doesn't just tell if there are more records to process, it also advances the cursor to the next row.

This means that even when you want to access only the first row of the result, you still have to call `next()` (preferably with an `if` statement, because if there are no elements, an `SQLException` is thrown if you try to access any):

```
if(rs.next) {
   // Access the first element if there's any
}
```

Now, to actually get the data, `ResultSet` has getter methods for a lot of data types, for example:

```
getInt() returns an int
getLong() returns a Long
getString() returns a String
getObject() returns an Object
getDate() returns a java.sql.Date
getTime() returns a java.sql.Time
getTimeStamp() returns java.sql.Timestamp
```

For each method, there are two versions:

- One that takes a `String` that represents the name of the column (this is **NOT** case-sensitive).
- Another one that takes an `int` that represents the column index according to the order declared in the `SELECT` clause. The first column starts with `1`, not `0`.

For example:

```
Result rs = stmt.executeQuery(
      "SELECT id, name FROM user"
);
while(rs.next()) {
   int id = rs.getInt("id");
   String name = rs.getString("name");
   // Do something
}
```

It's equivalent to:

```
Result rs = stmt.executeQuery(
      "SELECT id, name FROM user"
);
while(rs.next()) {
   int id = rs.getInt(1);
   String name = rs.getString(2);
   // Do something
}
```

If we reference a non-existent column (either by index or name), an `SQLException` will be thrown.

Notice that methods `getDate()`, `getTime()`, and `getTimeStamp()` don't return standard date or time objects, so they might need to be converted, for example:

```
Result rs = stmt.executeQuery(
      "SELECT insertion_date FROM user"
);
```

```
while(rs.next()) {
   // Getting the date part
   java.sql.Date sqlDate = rs.getDate(1);
   // Getting the time part
   java.sql.Time sqlTime = rs.getTime(1);
   // Getting both, the date and time part
   java.sql.Timestamp sqlTimestamp =
                        rs.getTimestamp(1);
   // Converting date
   LocalDate localDate = sqlDate.toLocalDate();
   // Converting time
   LocalTime localTime = sqlTime.toLocalTime();
   // Converting timestamp
   Instant instant = sqlTimestamp.toInstant();
   LocalDateTime localDateTime =
               sqlTimestamp.toLocalDateTime();
}
```

So that's how you work with `TYPE_FORWAD_ONLY` result sets.

When working with scrollable result sets (`TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`), we have a lot of options to move the cursor.

Here's the complete list of methods for moving the cursor (just remember that all methods, except `next()` , need a scrollable result set):

Method	Description
`boolean absolute(int row)`	Moves the cursor to the given row number in the result set, counting from the beginning (if the argument is positive) or the end (if negative). If the argument is zero, the cursor is moved before the first row. It returns true if the cursor is moved to a valid position or false if the cursor is before the first row or after the last row.
`void afterLast()`	Moves the cursor after the last row.
`void beforeFirst()`	Moves the cursor before the first row.
`boolean first()`	Moves the cursor to the first row. It returns true if the cursor is on a valid row or false if there are no rows in the result set.
`boolean last()`	Moves the cursor to the last row. Returns true if the cursor is on a valid row or false if there are no rows in the result set.
`boolean next()`	Moves the cursor to the next row. It returns true if the new current row is valid or false if there are no more rows.
`boolean previous()`	Moves the cursor to the previous row. It returns true if the new current row is valid or false if the cursor is before the first row.
`boolean relative(int rows)`	Moves the cursor a relative number of rows, either positive or negative. Moving beyond the first/last row in the result set positions the cursor before/after the first/last row. It returns true if the cursor is on a valid row, false otherwise.

So considering this table:

ID	NAME	INSERTION_DATE
1	THOMAS	2016 / 03 / 01

2	LAURA	2016 / 03 / 01
3	MAX	2016 / 03 / 01
4	KIM	2016 / 03 / 01

The following program shows some of these methods. Try to follow it:

```
try(Connection con =
         DriverManager.getConnection(
                                 url, user, passw);
         Statement stmt = con.createStatement(
                  ResultSet.TYPE_SCROLL_INSENSITIVE,
                  ResultSet.CONCUR_READ_ONLY);
         ResultSet rs = stmt.executeQuery(
                  "SELECT * FROM user")) {
   System.out.println(rs.absolute(3)); // true
   System.out.println(rs.getInt(1)); // 3
   System.out.println(rs.absolute(-3)); // true
   System.out.println(rs.getInt(1)); // 2
   System.out.println(rs.absolute(0)); // false
   System.out.println(rs.next()); // true
   System.out.println(rs.getInt(1)); // 1
   System.out.println(rs.previous()); // false
   System.out.println(rs.relative(2)); // true
   System.out.println(rs.getInt(1)); // 2
   System.out.println(rs.relative(0)); // true
   System.out.println(rs.getInt(1)); // 2
   System.out.println(rs.relative(10)); // false
   System.out.println(rs.previous()); // true
   System.out.println(rs.getInt(1)); // 4
} catch(SQLException e) {
   e.printStackTrace();
}
```

Key Points

- The Java Database Connectivity API (or JDBC) defines how we can access a relational database. You can find its classes and interfaces in the `java.sql` package.
- When using JDBC, you work with interfaces rather than implementations. Those implementations come from a JDBC driver. The four main interfaces to implement are:
  - `java.sql.Driver`
  - `java.sql.Connection`
  - `java.sql.Statement`
  - `java.sql.ResultSet`
- In addition to these interfaces, an important class is `java.sql.DriverManager`, which keeps track of the loaded JDBC drivers and gets the actual connection to the database.
- Once the drivers are loaded, you can connect to a database with the static method `DriverManager.getConnection(String url)`.
- The URL varies depending on the database used. However, they have three parts in common:
  - The first part is always the same, *jdbc*.
  - The second part, most of the time, is the name of the database and/or the type of the driver, like *mysql*, *postgresql*, *oracle:thin*.
  - The last part varies according to the database, but most of the time, it contains the name (or IP) of the host, the port and the name of the database you're connecting to, like *//192.168.0.1:3306/db1*.
- Once we have a `Connection` object, we can execute some SQL statements by getting a `Statement` object.

- You can get a `Statement` from a `Connection` object using the `createStatement()` method.
- When creating a `Statement`, you can define the type of the result set and its concurrency mode.
- There are three types of result sets, `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, `ResultSet.TYPE_SCROLL_SENSITIVE`.
- There are two concurrency modes, `ResultSet.CONCUR_READ_ONLY`, `ResultSet.CONCUR_UPDATABLE`.
- We have three methods in `Statement` to execute SQL commands:
  - `boolean execute(String sql)`
  - `ResultSet executeQuery(String sql)`
  - `int executeUpdate(String sql)`
- A `ResultSet` object is used to read the results of a query in a tabular format (rows containing the columns specified). This object keeps a cursor that points to the current row and you can only read one row at a time.
- The rules for closing JDBC resources are:
  - The `ResultSet` object is closed first, then the `Statement` object, then the `Connection` object.
  - A `ResultSet` is automatically closed when another `ResultSet` is executed from the same `Statement` object.
  - Closing a `Statement` also closes the `ResultSet`.
  - Closing a `Connection` also closes the `Statement` and `ResultSet` objects.
- To actually get the data, `ResultSet` has getter methods for a lot of data types. For each method, there are two versions, one that takes the name of the column and another that takes its column index.
- When working with scrollable result sets (`TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`), we have a lot of options to move the cursor, like `absolute(int row)`, `first()`, `previous()`, and `relative(int rows)`.

Self Test

1. Given:

```
public class Question_29_1 {
   public static void main(String[] args) {
      try(Connection con =
            DriverManager.getConnection(
                     "jdbc:mysql://localhost");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(
                     "SELECT * FROM user")) {
         while(rs.next()) {
            System.out.println(rs.getObject(1));
         }
      } catch(SQLException e) {
         System.out.println("SQLException");
      }
   }
}
```

What is the result if the query doesn't return any result?
A. `SQLException`
B. Nothing is printed
C. Compilation fails
D. An uncaught exception occurs at runtime

2. Which of the following is equivalent to `rs.absolute(-1)`?
A. `rs.absolute(1);`
B. `rs.afterLast();`

C. `rs.last();`
D. `rs.relative(-1);`

3. Which of the following options shows the correct order to close database resources?
A. `ResultSet , Connection , Statement`
B. `Statement , ResultSet , Connection`
C. `Connection , Statement , ResultSet`
D. `ResultSet , Statement , Connection`

4. Given:

```
public class Question_29_4 {
   public static void main(String[] args) {
      try(Connection con =
            DriverManager.getConnection(
                     "jdbc:mysql://localhost");
            Statement stmt = con.createStatement()) {
         System.out.println(
               stmt.execute(
                  "INSERT INTO user VALUES(1,'Joe')"
               )
         );
      } catch(SQLException e) { /** ... */ }
   }
}
```

What is the result?
A. `true`
B. `false`
C. `1`
D. An exception occurs at runtime

5. Which of the following can be a valid way to get the value of the first column of a row?
A. `rs.getInteger(1);`
B. `rb.getString("0");`
C. `rb.getObject(0);`
D. `rb.getBoolean(1);`

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

28. Fork/Join Framework 30. Localization